42392.P9770

PATENT

UNITED STATES DESIGN PATENT APPLICATION

FOR

NEW PROCESSOR MODE FOR LIMITING THE OPERATION OF GUEST SOFTWARE RUNNING ON A VIRTUAL MACHINE SUPPORTED BY A VIRTUAL MACHINE MONITOR

Inventors:

GILBERT NEIGER

STEPHEN T. CHOU

ERIK COTA-ROBLES

STALINSELVARAJ JEYASINGH

ALAIN KAGI

MICHAEL A. KOZUCH

RICHARD UHLIG

Prepared by:

BLAKELY SOKOLOFF TAYLOR & ZAFMAN LLP
12400 Wilshire Boulevard, Seventh Floor
Los Angeles, CA 90025-1026

(408) 720-8300

NEW PROCESSOR MODE FOR LIMITING THE OPERATION OF GUEST SOFTWARE RUNNING ON A VIRTUAL MACHINE SUPPORTED BY A VIRTUAL MACHINE MONITOR

Field of the Invention

The present invention relates generally to virtual machines, and more specifically to providing processor support for a virtual-machine monitor.

Background of the Invention

A conventional virtual-machine monitor (VMM) typically runs on a computer and presents to other software the abstraction of one or more virtual machines. Each virtual machine may function as a self-contained platform, running its own "guest operating system" (i.e., an operating system hosted by the VMM). The guest operating system expects to operate as if it were running on a dedicated computer rather than a virtual machine. That is, the guest operating system expects to control various computer operations and have access to hardware resources during these operations. The hardware resources may include processor-resident resources (e.g., control registers) and resources that reside in memory (e.g., descriptor tables). However, in a virtual-machine environment, the VMM should be able to have ultimate control over these resources to provide proper operation of virtual machines and protection from and between virtual machines. To achieve this, the VMM typically intercepts and arbitrates all accesses made by the guest operating system to the hardware resources.

Current implementations of VMMs may be based on software techniques for controlling access to hardware resources by the guest operating system. However, these software techniques may lack the ability to prevent guest software from accessing some fields in the processor's control registers and memory. For instance, the guest operating system may not be prevented from accessing a requestor privilege level (RPL) field in the code segment register of IA-32 microprocessors. In addition, existing software techniques typically suffer from performance problems. Thus, an alternative mechanism is needed for supporting the operation of the VMM.

Brief Description of the Drawings

The present invention is illustrated by way of example, and not by way of limitation, in the figures of the accompanying drawings and in which like reference numerals refer to similar elements and in which:

Figure 1 illustrates one embodiment of a virtual-machine environment;

Figure 2 illustrates operation of a virtual-machine monitor based on guest deprivileging;

Figure 3 is a block diagram of a system for providing processor support to a virtual-machine monitor, according to one embodiment of the present invention;

Figure 4 is a flow diagram of a method for providing processor support to a virtual-machine monitor, according to one embodiment of the present invention;

Figure 5 is a flow diagram of a method for performing a transition out of V32 mode, according to one embodiment of the present invention;

Figure 6 is a flow diagram of a method for generating virtualization traps, according to one embodiment of the present invention;

Figure 7 is a flow diagram of a method for maintaining a redirection map, according to one embodiment of the present invention;

Figure 8 is a flow diagram of a method for controlling masking of interrupts, according to one embodiment of the present invention; and

Figure 9 is a block diagram of one embodiment of a processing system.

Description of Embodiments

A method and apparatus for providing processor support to a virtual-machine monitor are described. In the following description, for purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art that the present invention can be practiced without these specific details.

Some portions of the detailed descriptions that follow are presented in terms of algorithms and symbolic representations of operations on data bits within a computer memory. These algorithmic descriptions and representations are the means used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. An algorithm is here, and generally, conceived to be a self-consistent sequence of steps leading to a desired result. The steps are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like.

It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically stated otherwise as apparent from the following discussions, it is appreciated that

throughout the present invention, discussions utilizing terms such as "processing" or "computing" or "calculating" or "determining" or "displaying" or the like, may refer to the action and processes of a computer system, or similar electronic computing device, that manipulates and transforms data represented as physical (electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the computer-system memories or registers or other such information storage, transmission or display devices.

The present invention also relates to apparatus for performing the operations herein. This apparatus may be specially constructed for the required purposes, or it may comprise a general purpose computer selectively activated or reconfigured by a computer program stored in the computer. Such a computer program may be stored in a computer readable storage medium, such as, but is not limited to, any type of disk including floppy disks, optical disks, CD-ROMs, and magnetic-optical disks, read-only memories (ROMs), random access memories (RAMs), EPROMs, EEPROMs, magnetic or optical cards, or any type of media suitable for storing electronic instructions, and each coupled to a computer system bus. Instructions are executable using one or more processing devices (e.g., processors, central processing units, etc.).

The algorithms and displays presented herein are not inherently related to any particular computer or other apparatus. Various general-purpose machines may be used with programs in accordance with the teachings herein, or it may prove convenient to construct more specialized

apparatus to perform the required method steps. The required structure for a variety of these machines will appear from the description below. In addition, the present invention is not described with reference to any particular programming language. It will be appreciated that a variety of programming languages may be used to implement the teachings of the invention as described herein.

In the following detailed description of the embodiments, reference is made to the accompanying drawings that show, by way of illustration, specific embodiments in which the invention may be practiced. In the drawings, like numerals describe substantially similar components throughout the several views. These embodiments are described in sufficient detail to enable those skilled in the art to practice the invention. Other embodiments may be utilized and structural, logical, and electrical changes may be made without departing from the scope of the present invention. Moreover, it is to be understood that the various embodiments of the invention, although different, are not necessarily mutually exclusive. For example, a particular feature, structure, or characteristic described in one embodiment may be included within other embodiments. The following detailed description is, therefore, not to be taken in a limiting sense, and the scope of the present invention is defined only by the appended claims, along with the full scope of equivalents to which such claims are entitled.

The method and apparatus of the present invention provide processor support for a virtual-machine monitor (VMM). **Figure 1** illustrates one embodiment of a virtual-machine environment 100, in which the present

invention may operate. In this embodiment, bare platform hardware 116 comprises a computing platform, which may be capable, for example, of executing a standard operating system (OS) or a virtual-machine monitor (VMM), such as a VMM 112. A VMM, though typically implemented in software, may export a bare machine interface, such as an emulation, to higher level software. Such higher level software may comprise a standard or real-time OS, although the invention is not limited in scope in this respect and, alternatively, for example, a VMM may be run within, or on top of, another VMM. VMMs and their typical features and functionality are well-known by those skilled in the art and may be implemented, for example, in software, firmware or by a combination of various techniques.

As described above, a VMM presents to other software (i.e., "guest" software) the abstraction of one or more virtual machines (VMs). **Figure 1** shows two VMs, 102 and 114. The guest software of each VM includes a guest OS such as a guest OS 104 or 106 and various guest software applications 108-110. Each of the guest OSs 104 and 106 expects to control access to physical resources (e.g., processor registers, memory and memory-mapped I/O devices) within the hardware platform on which the guest OS 104 or 106 is running and to perform other functions. However, in a virtual-machine environment, the VMM 112 should be able to have ultimate control over the physical resources to provide proper operation of VMs 102 and 112 and protection from and between VMs 102 and 114. The VMM 112 achieves this goal by intercepting all accesses of the guest OSs 104 and 106 to the computer's physical resources. Various techniques may be used to enable the

VMM 112 to intercept the above accesses. One of such techniques is a guest-deprivileging technique which forces all guest software to run at a hardware privilege level that does not allow that software access to certain hardware resources. As a result, whenever the guest OS 104 or 106 attempts to access any of these hardware resources, it "traps" to the VMM 112, i.e., the VMM 112 receives control over an operation initiated by the guest OS if this operation involves accessing such hardware resources.

Figure 2 illustrates a prior art embodiment of the operation of a VMM that supports guest deprivileging. As described above, guest deprivileging forces a guest OS to execute in a less privileged mode of execution. For IA-32 microprocessors, the nature of page-based protection is such that all guest software runs at the least privileged level (i.e., ring 3). That is, a guest OS 206 and guest applications 204 run at the same privilege level. As a result, the guest OS 206 may not be able to protect itself from the guest applications 206, thereby possibly compromising the integrity of the guest OS 206. This problem is known as ring compression.

Guest deprivileging may also cause an address-space compression problem. As described above, certain attempts of guest software to access hardware resources result in traps that transfer control to the VMM 220. In order to enable this transfer of control, a portion of VMM code and/or data structures may be architecturally required to reside in the same virtual-address space as the guest OS 206. For instance, the IA-32 instruction-set architecture (ISA) may require that an interrupt descriptor table (IDT) 212, a global descriptor table (GDT) 210 and trap handling routines reside at the

same virtual space as the guest OS 206. The VMM code and data structures 220 that reside in the virtual space 202 must be protected from accesses by guest software (e.g., by running at ring 0). Accordingly, the guest OS 206 does not control the entire address space 202 as the guest OS 206 expects. This causes an address-space compression problem.

Another limitation of VMMs that use guest deprivileging pertains to some cases in which the processors fail to prevent guest software from reading privileged hardware resources. For instance, the IA-32 microprocessors allow the guest OS 206 to execute PUSH CS instructions which store a code segment register into memory. One of this register's fields stores information about the current privilege level. Accordingly, the guest OS 206 can become aware that its privilege level is 3, and not 0 as the guest OS 206 expects, by reading the value of the current privilege level from the memory. As a result, the guest OS 206 may be exposed to the fact that it is running on a virtual machine, and the integrity of the guest OS 206 may be compromised.

Similarly, in some cases, the processors do not trap an attempt of the guest software to modify privileged software resources. For instance, the IA-32 processors allow the guest OS 206 to issue POPF instructions which attempt to load EFLAGS, and instead of generating a trap, simply ignore all or part of such attempts of the guest OS 206 because the guest OS 206 executes these instructions with insufficient privilege. As a result, the guest OS 206 believes that a corresponding EFLAGS field has been modified but the VMM 220 is not aware of that and cannot properly emulate this modification. Accordingly,

the guest OS 206 may be exposed to the fact that it is running on a virtual machine, and the integrity of the guest OS 206 may be compromised.

Yet another limitation of VM monitors that use guest deprivileging is caused by excessive trapping. Because the number of hardware resource elements that need to be protected from accesses by guest software is significant and such accesses may be frequent, traps may occur often. For instance, the IA-32 microprocessors support CLI instructions. The CLI instructions are issued to modify an interrupt flag, which is an element of the privileged hardware resources and which thus cannot be accessed by unprivileged software. The guest OS 206 commonly issues these instructions during its operation, thereby causing frequent traps to the VMM 220. Frequent trapping negatively affects system performance and reduces the utility of the VMM 220.

The present invention addresses the above problems and various other limitations by providing processor support for a VMM. **Figure 3** is block diagram of a system for providing processor support to a virtual-machine monitor, according to one embodiment of the present invention.

Referring to **Figure 3**, all guest software runs at a processor mode referred to herein as a virtual 32-bit mode (V32 mode). V32 mode allows the guest software to run at its intended privilege level. For instance, for the IA-32 ISA, the guest OS 308 runs at the most privileged level (i.e., ring 0) and guest applications 306 run at the least privileged level (i.e., ring 3). V32 mode restricts the operation of the guest software by preventing the guest software from performing operations that may result in its access of certain privileged

hardware resources. V32 mode is exited when the guest software attempts to perform such an operation.

The VMM 320 runs outside V32 mode. When a transition out of V32 mode occurs, the VMM 320 receives control over the operation initiated by the guest OS 308 or guest application 306. The VMM 320 then performs this operation, and transfers control back to the guest software by entering V32 mode, thereby emulating the functionality desired by the guest software.

In one embodiment, V32 mode is implemented by maintaining a flag in one of the processor's control registers (e.g., CR0) to indicate whether the processor is in V32 mode or not. In another embodiment, this flag (referred to herein as EFLAGS.V32) is maintained in one of the reserved bits in the upper half of EFLAGS. The EFLAGS.V32 flag is modified either by a transition out of V32 mode or a transition into V32 mode.

In one embodiment, the ability of the processor to support V32 mode are reported using one of the reserved feature bits that are returned in EDX when the CPUID instruction is executed with the value 1 in EAX. It should be noted that a variety of other mechanisms can be used to implement V32 mode and to report the ability of the processor to support V32 mode without loss of generality.

In one embodiment, certain exceptions and interrupts cause a transition out of V32 mode. These exceptions and interrupts include "virtualization traps." A virtualization trap is generated when guest software that runs in V32 mode attempts to perform an operation that may result in its access of certain privileged hardware resources. In one embodiment, when a transition

out of V32 mode occurs, the guest address space 304 is automatically changed to the VMM address space 302. In addition, the processor state that was used by guest software is saved and stored in temporary registers, and the processor state required by the VMM 320 is loaded.

In one embodiment, when a transition into V32 mode occurs, the processor state that was saved on the transition out of V32 mode (i.e., to the VMM 320) is automatically restored, the VMM address space 302 is changed to the guest address space 304, and control is returned to the guest OS 308.

In one embodiment, when guest software runs in V32 mode, software interrupts (e.g., interrupts caused by execution of BOUND, INT or INTO instructions) are handled by the guest OS 308 using the guest IDT (i.e., the IDT residing in the guest address space 304). All other interrupts and exceptions including virtualization traps cause a transition out of V32 mode which results in a change of the guest address space 304 to the VMM address space 302. The IDT 316 is then used to point to code that handles a corresponding exception or interrupt.

In one embodiment, a new interrupt flag (i.e., a virtual-machine interrupt flag) is maintained for accesses by guest software. Whenever guest software attempts to access the interrupt flag (IF), it will instead access the virtual machine interrupt flag (VMIF). In one embodiment, an attempt of guest software to access VMIF (e.g., using the CLI instruction) does not cause a transition out of V32 mode, except when the guest OS 308 has just set VMIF to 1 (e.g., through the STI instruction) and the VMM 320 wishes to deliver a pending interrupt to the guest OS 308. Such pending interrupts referred to

herein as "virtual pending interrupts" generate virtualization traps which allow the VMM 320 to deliver a pending interrupt to the guest software when the guest OS 308 signals that it is ready to process such an interrupt. In one embodiment, one of the reserved bits in the upper half of the EFLAGS register is used to maintain a flag indicating whether guest software has a pending virtual interrupt.

The implementation of V32 mode allows resolving all of the problems caused guest deprivileging as described above. In particular, because guest software runs in V32 mode at its intended privilege level, the problem of ring compression is eliminated. In addition, address-space compression is no longer a problem because a virtualization trap automatically causes a switch to the VMM address space 302, and therefore neither the tables controlling such transfers nor the code handling a corresponding virtualization trap is required to reside in the guest address space 304.

Furthermore, because V32 mode enables the guest software to run at its intended privilege level, the hardware resources that need to be protected no longer include those elements of hardware resources that control the privilege level. For instance, the PUSH CS instruction described above can no longer reveal to the guest OS 308 that it runs on a virtual machine because the field of the code segment register that stores information about a current privilege level now stores the privilege level intended by the guest OS 308. Similarly, POPF instructions which attempt to load EFLAGS are no longer ignored when executed by the guest OS 308 because the guest OS 206 executes these instructions with sufficient privilege.

Accordingly, the number of elements of hardware resources that need to be protected is reduced. If any of them allow non-trapping read or write accesses by guest software, they are specifically architected to cause traps when executed in V32 mode. Thus, the problems caused by non-trapping read and write accesses are eliminated. In addition, because the implementation of V32 mode reduces the number of elements of hardware resources that need to be protected, the number of traps that occur when guest software attempts to access these elements is also reduced. Frequency of traps is further reduced by providing mechanisms for eliminating traps caused by the most frequently used instructions. For instance, STI instructions no longer cause traps except when guest software has a pending virtual interrupt.

Figure 4 is a flow diagram of a method 400 for providing processor support to a virtual machine monitor, according to one embodiment of the present invention. At processing block 404, guest software is executed in a processor mode (i.e., V32 mode) that allows guest software to operate at a privilege level intended by the guest software. That is, a guest OS may operate at a supervisor privilege level, and guest applications may operate at a user privilege level.

At processing block 406, an attempt of the guest software to perform an operation restricted by V32 mode is identified. In response to this attempt, V32 mode is exited to transfer control over the operation initiated by the guest software to the VMM which runs outside V32 mode (processor block 408). In one embodiment, the VMM configures what operations should cause a

transition out of V32 mode as will be described in greater detail below in conjunction with **Figure 7**. In one embodiment, such operations generate virtualization traps that cause a transition out of V32 mode. Alternatively, any other mechanism known in the art can be used to cause a transition out of V32 mode. One embodiment of performing a transition out of V32 mode is described in greater detail below in conjunction with **Figure 5**.

Further, the VMM responds to the operation intended by the guest software (processing block 410). Afterwards, V32 mode is re-entered to transfer control over this operation back to the guest software (processing block 412), and method 400 returns to processing block 404. In one embodiment, when a transition into V32 mode occurs, the processor state expected by the guest software is automatically restored and the VMM address space is changed to the guest address space.

Figure 5 is a flow diagram of a method 500 for performing a transition out of V32 mode, according to one embodiment of the present invention. Method 500 begins with saving processor state used by guest software (processing block 504). In one embodiment, the saved processor stated is stored in the processor's temporary registers. At processing block 506, the processor state required by the VMM is loaded into processor registers. In one embodiment, loading the processor state affects a change of the guest address space to the VMM address space (e.g., the processor state is loaded by loading the control register CR3). In an alternative embodiment, loading the processor state does not cause a change in the address space. In such an embodiment, at processing block 508, an address space switch is performed to transfer from

the guest address space to the VMM address space. Accordingly, when an interrupt or exception causing the transition occurs, the IDT residing in the VMM address space is automatically used to point to the VMM-resident code for handling this interrupt or exception. **Figure 6** is a flow diagram of a method 600 for generating virtualization traps, according to one embodiment of the present invention. Method 600 begins with identifying an attempt of guest software to perform an operation that may be restricted by V32 mode (processing block 604). At decision box 606, a determination is made as to whether the attempt of the guest software can potentially succeed. If the determination is positive, a virtualization trap is generated (processing block 608). Alternatively, no virtualization trap is generated, and the guest software proceeds with the operation (processing block 610). For instance, according to the IA-32 ISA, the RDMSR instruction can be executed only by software running with supervisor privilege. Consequently, if the guest software OS which runs with supervisor privilege executes this instruction, its attempt may be successful. If a guest application which runs with user privilege executes this instruction, its attempt will not be successful, and a general-protection fault will occur. Accordingly, an attempt of the guest OS to execute the RDMSR instruction will cause a virtualization trap but an attempt of a guest application will be handled by the guest OS.

In one embodiment, virtualization traps will be caused by potentially successful attempts of the guest OS to access the processor's control registers (e.g., CR0-CR4). For instance, for IA-32 processors, virtualization traps will be generated in response to an attempt of the guest software to execute MOV CR

(except the attempts to store CR2, which do not need to cause virtualization traps), CLTS, LMSW or SMSW instructions, or a task switch. Virtualization traps may be also caused by a potentially successful attempt of the guest software to set an interrupt flag IF (e.g., via STI, POPF or IRET instructions) if guest software has a pending virtual interrupt. For IA-32 ISA, successful attempts to execute such instructions as, for example, HLT, IN, INS/INSB/INSW/INSD, INVD, OUT, OUTS/OUTSB/OUTSW/OUTSD, RDMSR, and WRMSR, will cause virtualization traps. These virtualization traps will prevent guest software from halting the processor and from directly accessing I/O ports, caches or model-specific registers. In addition, virtualization traps may be caused by attempts to execute CPUID instructions to allow the VMM to present the abstraction of processor features chosen by the VMM, by attempts to execute INVLPG instructions to enable the VMM to properly virtualize address translations, and by attempts to execute IRET instructions (if IRET is used to transition into V32 mode) used by guest software to implement a VMM to allow recursively nested VMMs.

Figure 7 is a flow diagram of a method 700 for maintaining a redirection map, according to one embodiment of the present invention. According to this embodiment, the VMM maintains a redirection map to configure which interrupts and exceptions should result in a virtualization trap (processing block 704). At processing block 706, an occurrence of an interrupt or exception is identified. The redirection map is then consulted to find a bit associated with this interrupt or exception in the redirection bitmap (processing block 708).

At decision box 710, a determination is made as to whether this interrupt is allowed to be handled by the guest OS. If the determination is positive, the interrupt or exception is delivered to V32 mode and is handled by the guest OS (processing block 714). Alternatively, a virtualization trap is generated, causing a transition out of V32 mode (processing block 712).

Figure 8 is a flow diagram of a method 800 for controlling masking of interrupts, according to one embodiment of the present invention. Various embodiments may be used to control the masking of interrupts. In one embodiment, all interrupts are unmasked when guest software is running. In this embodiment, the guest software is permitted to manipulate an interrupt flag (e.g., for IA-32 microprocessors, this flag is identified as EFLAGS.IF), but this manipulation will be ignored with respect to the masking of interrupts. In another embodiment, the masking of interrupts is dependent on the interrupt flag. In this embodiment, the guest software is not permitted to manipulate the interrupt flag. In particular, the guest software may be prevented from accessing the interrupt flag by providing a shadow interrupt flag (e.g., EFLAGS.VMIF) for modifications by the guest software, by generating a virtualization trap in response to such an attempt of the guest software, or by using any other technique known in the art.

Method 800 begins with identifying an attempt of guest software to modify an interrupt flag that may potentially control masking of interrupts (processing block 804). At decision box 806, a determination is made as to whether the interrupt flag controls the masking of interrupts. If the determination is negative, i.e., all interrupts are unmasked, the guest software

is allowed to modify the interrupt flag (processing block 808). As described above, this modification will not have an effect on the masking of the interrupts.

Otherwise, if the masking of interrupts is dependent on the interrupt flag, a determination is then made as to whether a shadow interrupt flag exists, i.e., whether the attempt of the guest software to affect the masking of interrupts is affecting the shadow flag (decision box 810). If the determination is negative, i.e., the guest software attempts to modify the actual interrupt flag, a virtualization trap occurs (processing block 812), causing a transition out of V32 mode (processing block 816). Alternatively, if the actual interrupt flag is not accessible to the guest software, the guest software is allowed to modify the shadow interrupt flag (processing block 814).

Figure 9 is a block diagram of one embodiment of a processing system. Processing system 900 includes processor 920 and memory 930. Processor 920 can be any type of processor capable of executing software, such as a microprocessor, digital signal processor, microcontroller, or the like. Processing system 900 can be a personal computer (PC), mainframe, handheld device, portable computer, set-top box, or any other system that includes software.

Memory 930 can be a hard disk, a floppy disk, random access memory (RAM), read only memory (ROM), flash memory, or any other type of machine medium readable by processor 920. Memory 930 can store instructions for performing the execution of the various method embodiments

of the present invention such as methods 400, 500, 600, 700 and 800 (**Figures 4-8**).

It is to be understood that the above description is intended to be illustrative, and not restrictive. Many other embodiments will be apparent to those of skill in the art upon reading and understanding the above description. The scope of the invention should, therefore, be determined with reference to the appended claims, along with the full scope of equivalents to which such claims are entitled.